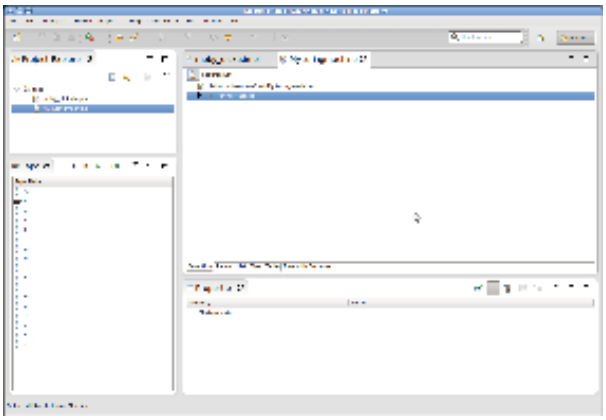
The Eclipse Modeling Framework

This tutorial will be all about the Eclipse Modeling Framework, one of the core technologies in the Eclipse universe. You will learn what metamodels are and how to create them, how to generate an editor for instances of your metamodels, and how to load and save such instances. We will of course continue our example from the last tutorial: the metamodel you will create will be one for models that specify Turing Machines. To learn about using EMF models programmatically, you will also implement a head controller that executes a Turing Machine given to it in terms of a Turing Machine model.

Once you're done with this tutorial, you will have an application that looks something like this:



You may want to download the slides of the presentation explaining the basic concepts you will explore in this tutorial.

Preliminaries

The Eclipse Modeling Framework (EMF) belongs to the most important technologies in the Eclipse world. One of the more recent indicators of this is its being used to describe the very foundation of Eclipse 4 (or *e4*, which of course sounds way cooler) applications: the application model. We might come back to the application model in the project phase of our practical. Another indication is that EMF has come to be the foundation of most other modeling-related Eclipse projects. Thus, there is important stuff to be learned in this tutorial!

Let's try to get a first idea of what EMF does. Essentially, EMF helps you to write the code necessary to represent models (or domain models, as they are also called). So, what are models? An example of a model that you have all encountered before is a UML class diagram. In such a model, classes are related to one another using different types of relations: inheritance relations, associations, aggregations, compositions,... Working on such a model requires code that represents the model and that allows you to change its structure. EMF can help you write that code.

However, EMF can do that for all kinds of models. To generate the code for a specific kind of models, EMF needs to know what kinds of entities and relationships between the entities there are in those models. This is where a metamodel comes into play: metamodels describe the structure of your models. They are to your models what grammar is to your programming languages. When working with EMF, you usually start by defining the metamodel (syntax of your models), and then proceed to let EMF generate the code required to represent such models.

You can also have EMF generate simple editors to load, edit, and save models corresponding to your metamodel.

EMF also contains a bunch of tools that can help you with everything surrounding modeling, such as loading and saving models. EMF usually serializes models using the XMI (XML Metadata Interchange) format, which is an XML file geared towards the representation of models. The classes generated by EMF also have built-in facilities to, for instance, help you observe changes in the model instances.

Required Software

Your Eclipse installation already has everything we need for this tutorial. In fact, that's one reason why we suggested you install the Eclipse Modeling Tools. You will again be working on your branch of our tutorials Git repository.

Finding Documentation

For an introduction to EMF, here's a few suggestions to get you started:

- The Wikipedia article on metamodeling may be i
- The Eclipse online help system contains a section on EMF, complete with an introduction and tutorials.
- There is a book on EMF, which is a great resource. The library may have copies available. We also have at least one copy at our office, so feel free to drop in and read it. (We also have tea and a sofa, so there's no lack of proper reading atmosphere... 😉)
- Among many other helpful tutorials, Lars Vogel, an active Eclipse developer, has also written a tutorial on EMF.

Creating a Metamodel

As we have already seen, everything in EMF begins with the metamodel. Metamodels can be specified in different formats: XSD (XML Schema Definition), UML, Ecore models,... In this tutorial, we will be using Ecore models to specify our metamodels. Take a moment to think about this: we're creating an Ecore model by drawing a diagram, and this model will then be used as the metamodel for our Turing Machine models. So, let's start by creating an Ecore diagram.

1. Create a new *Empty EMF Project* named `de.cau.cs.rtprak.login.turingmodel`. Remember to create the project in your Git repository. Once you click the *Finish* button, the *Empty EMF Project* wizard creates a new plug-in project for you, complete with a `src` folder for Java source files, the `MANIFEST.MF` file we have encountered before, and, most importantly, a models folder that you will store your modeling files in. If you open the manifest file in the *Plugin Manifest Editor*, you will see that the wizard already added a dependency to `org.eclipse.emf.ecore`, which all EMF projects depend on.
2. Create a new Ecore diagram in the models folder by right-clicking the folder and clicking *New -> Other...*, and then selecting *Ecore Diagram* from the *Ecore Tools* category. Note that the category *Other* also contains an entry *Ecore Diagram*. However, the editor we will be using has more features and is more user-friendly than the one in the *Other* category.
3. In the *New Ecore Diagram* wizard, check *Create a new model* and choose `turingmachine.ecore` as the *Domain file name*. Once you click *Finish*, the wizard will generate two files for you:
    - `turingmachine.ecore` contains the information about the data structures in your Turing Machine models. In effect, it is your metamodel in Ecore format.
    - `turingmachine.ecorediag` is the diagram you're editing and contains things like coordinates of the different data structures, and bend points of the relations between them – in short, everything the graphical editor needs to know to display the diagram.
4. You will need the *Properties* view to edit your model properly. This view shows detailed information about the currently selected model element and lets you edit them. It also shows general information about the model if no specific element it selected. Summon the *Properties* view now by right-clicking into your diagram and selecting *Show Properties View*.
5. Now that the *Properties* view is visible, switch to its *Model* tab and set the following properties:
    - Name: Model elements are grouped into packages, and this is the package name. Set to `turingmachine`.
    - Ns Prefix: Namespace prefix that will be used in the XML representation of your models later on. Use something short, e.g. `turing`.
    - Ns URI: While the package name need not be unique, namespace URI's are used to uniquely identify stuff. The usual convention is to use a name following the format `http://project_name_part/packagename`. Thus, set this to something like `http://de.cau.cs.rtprak.login/turingmachine`.

Modeling Your Turing Machines

Now that we have an empty Ecore diagram it's time to get to the interesting part: defining the metamodel for your Turing Machines. You will later write a simulator that will execute Turing Machines specified as models following this metamodel; keep that in mind while designing the metamodel. This is a complex and interesting task that will require some thought on your part. Feel free to discuss this with other participants: talking about a problem with other people usually leads to better designs and helps you think about problems that you might have overlooked otherwise. Here's some first suggestions for design decisions you're facing to get you started:

- Do you model the state machine? If so, what information do you need to specify states and transitions?
- How will your simulator know which state to start in?
- Do you model the Turing Machine's tape?

Model Elements

You will need the following Ecore model elements:

- *EClass* – Create one for every item that you want to be in your model. Make sure that you have exactly one root element, that is, an element that represents your Turing Machine (perhaps `TuringMachine` would actually be a good name for it...) and provides access to other elements.
- *EAttribute* – Add attributes to classes to give them properties, e.g. a `name` attribute for states in state machines. The most important property of attributes is their type, which you can configure in the *Properties* view.
- *EEnum* – Create enumerations to define simple enumeration types that you can then use as the type of attributes. Each item of the enumeration is an *EEnumLiteral*.
- *Inheritance Relations* – Use these as you would in UML class diagrams or ye plain ol' Java.
- *EReference* – Use references to provide links between classes. Here's a few things about references:
    - Every class (except the root class) requires exactly one *Containment* reference that specifies where it belongs to and where it will be stored later on when you save your models to XML files.
    - Set lower and upper bounds on references to control how many instances of a class can be referenced (just like multiplicities in UML class diagram associations).

- Consider whether a reference should have an opposite reference: a second reference in the other direction to be able to navigate back and forth between the model objects. Let's take two classes as an example to illustrate this: `Parent` and `Child`, where `Parent` can reference multiple `Child` objects. To be able to ask the `Parent` about all its children, we would add a reference `children` from `Parent` to `Child` with the containment flag active (that is, `Child` is part of its `Parent`). To be able to ask a `Child` about its `Parent`, we would add a second reference from `Child` to `Parent` with the *EOpposite* set to the `children` reference.

For this task, you won't need any more model elements.

One last thing before you get started: While working on your model, save and validate it regularly (*Edit -> Validate*). This will help you find potential problems with your model while you're still able to fix them easily.

Code Generation

In order to work with the data structures you specified in your metamodel conveniently, the model must be translated into Java code. For this purpose EMF provides a Java code generation facility that takes the metamodel as its input. Hence, you need a metamodel specified in one of the supported formats. Luckily, Ecore models are a supported format...

The Ecore model, however, is not sufficient to generate the code since it does not contain any information about, e.g., where the code is to be generated. Therefore EMF uses *generator models* to preserve such information. Here, we will create such a model from our Ecore model automatically.

Generating a Generator Model

1. Add a new *EMF Generator Model* to the `model` folder. Give it the same name as the Ecore model, but choose `.genmodel` as the extension. Choose *Ecore model* as the importer and select your Ecore model file. Make sure your `turingmachine` package is selected and click *Finish*. You get a new model file, which is mainly a copy of the Ecore model augmented with additional information required for code generation. Note: The generator model file is now synchronized to the Ecore model file, which in turn is synchronized to the Ecore diagram file. Once the generator model is created, changes in the diagram will also be applied to the generator model, removing the need for you to regenerate the generator model every time you change something in the original model.
2. Open the generator model, select your package, and open the *Properties* view.
3. Enter a meaningful *Base Package*, that is, the package where the generated Java classes will be placed in. A meaningful package name would be the project name.

Generating the Code

This is a very laborious task:

1. Right-click the root node in the tree editor and click *Generate All*.
2. Wait.

EMF should now have generated (automatically and hopefully without errors) code in folders and even new projects for you:

- Folder `src` of the original project: This is the Java implementation of your metamodel. The most important code. It allows to instantiate your model.
- Project ...edit: This project contains helper classes for eclipse to present model elements in Eclipse widgets like tables, trees, etc.
- Project ...editor: This is a ready-to-use tree editor for model instances of your metamodel. (Nothing graphical though...)
- Project ...tets: This is a set of skeletons for JUnit tests for your model. However, you have to implement the test methods yourself. We won't use it now, but feel free to have a look at it.

Note that the generated code is *not synchronized* to your models. If you make changes to your models, you have to re-generate your code. If you don't take care, manual changes in the generated code will get lost.

Launching the Tree Editor

You will of course be anxious to try out your new tree editor:

1. Launch an Eclipse application with all your workspace plug-ins.
2. In the new Eclipse instance, create a new empty project.
3. Create a new model instance (*File -> New -> Other -> Example EMF Model Creation Wizards -> Turingmachine Model*), choosing your root element as the model object.
4. Edit the model with the tree editor. Use the context menu to create new child elements, and the *Properties* view to configure elements.
5. Save the model.
6. Open the file with a text editor (your simple text editor, for example: right-click the model file and use the *Open with* menu).
7. To prepare for the next task, copy the model file you just created to your original developer workspace, into the root folder of the project where your model files are stored.

Saving and Loading Models

We will now look at working with EMF models, editing them through Java code instead of the GUI. We will also load and save them from / to XMI resources.

Creating a Test Project

We need a project to test with. If you already know your way around JUnit tests, you can skip creating a test project and just use the generated test project. If not, do the following:

1. Create a new empty *Plug-In Project*.
2. Add your metamodel project as a dependency of your new project through the *Plugin Manifest Editor*.
3. Create a simple Java class that implements a main method. Hint: In a new Java class, simply type main and hit Ctrl+Space. Eclipse content assist will create the method for you.
4. Import all packages of your metamodel code (i.e., `packagename`, `packagename.impl`, and `packagename.util`).

Creating a Model

> **Note**
>
> To talk about programmatically handling models, we will have to assume some sort of design for your model. The design we're assuming here is not the only possible design for your Turing Machines. Don't be alarmed if your model is different.

For instantiation of a model from code you cannot directly use the Java classes generated for the model. Instead, the main package contains interfaces for all of your model object classes. The `impl` package contains the actual implementation and the `util` package contains some helper classes. Do not instantiate objects directly by manually calling `new`. EMF generates a Factory to create new objects. The factory itself uses the singleton pattern to get access to it:

```
// assuming the model is called "Turing"
// and classes are "Model","State" and "Transition"
TuringFactory factory = TuringFactory.eINSTANCE;
Model myModel = factory.createModel();
State state1 = factory.createState();
State state2 = factory.createState();
Transition trans1 = factory.createTransition();
```

For all simple attributes, there are getter and setter methods:

```
state1.setName("State 1");
```

Simple references (multiplicity of 1) also have getters and setters:

```
// assume a transition has simple references to its source and target state
trans1.setSourceState(state1);
trans1.setTargetState(state2);
```

List references (multiplicity of > 1) have only a list getter, which is used to manipulate the list:

```
EList<State> states = myModel.getStates();
states.add(state1);
states.add(state2);
```

With these information out of the way, on we go to some model creation:

1. Create a valid model in the `main()` method with at least 5 Objects.
2. Run the `main()` method by right-clicking its class and selecting *Run as -> Java Application*. Note that this runs your `main()` method as a simple Java program, not a complete Eclipse application. EMF models can be used in any simple Java context, not just in Eclipse applications.

Saving a Model

EMF uses the Eclipse Resource concept to save models to files and load models from files. It can use different *Resource Factories* that determine how exactly models are serialized. We will use the XMIResourceFactoryImpl to save our models to XML files:

1. Add a dependency to the `org.eclipse.emf.ecore.xmi` plug-in.
2. Use something like the following code to save the model from above:

```
// Create a resource set.
ResourceSet resourceSet = new ResourceSetImpl();
```

```
// Register the default resource factory -- only needed for stand-alone!
// this tells EMF to use XML to save the model
resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put(
    Resource.Factory.Registry.DEFAULT_EXTENSION, new XMIResourceFactoryImpl());

// Get the URI of the model file.
URI fileURI = URI.createFileURI(new File("myTuringMachine.xmi").getAbsolutePath());

// Create a resource for this file.
Resource resource = resourceSet.createResource(fileURI);

// Add the model objects to the contents.
resource.getContents().add(myModel);

// Save the contents of the resource to the file system.
try
{
    resource.save(Collections.EMPTY_MAP); // the map can pass special saving options to the operation
} catch (IOException e) {
    /* error handling */
}
```

3. Execute the main method.
4. Refresh the project.
5. Hopefully be pleased about the new file in your project. Open the file to see if it contains all elements. Do you understand why opening the model does not correctly open your tree editor? Think about it. (And open it with a text editor.)

Loading a Model

1. Create a second class with a main() method.
2. Load the resource with something like the following code:

```
// Create a resource set.
ResourceSet resourceSet = new ResourceSetImpl();

// Register the default resource factory -- only needed for stand-alone!
resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put(
    Resource.Factory.Registry.DEFAULT_EXTENSION, new XMIResourceFactoryImpl());

// Register the package -- only needed for stand-alone!
// You find the correct name of the package in the generated model code
TuringPackage libraryPackage = TuringPackage.eINSTANCE;

// Get the URI of the model file.
URI fileURI = URI.createFileURI(new File("myTuringMachine.xmi").getAbsolutePath());

// Demand load the resource for this file, here the actual loading is done.
Resource resource = resourceSet.getResource(fileURI, true);

// get model elements from the resource
// note: get(0) might be dangerous. why?
EObject myModelObject = resource.getContents().get(0);

// Do something with the model
if (myModelObject instanceof Model) {
   // Model is the root class of your model
   for(State state: ((Model) myModelObject).getStates()){
      System.out.println(state.getName());
   }
}
```

3. Print the model to the console with something like the following code:

```
resource.save(System.out, Collections.EMPTY_MAP);
```

Implementing a Head Controller

This is where all the model design and model generation pays off and where we will establish the link to the second tutorial: you will write an `IHeadController` implementation that can simulate Turing Machines specified as models following the metamodel you just designed. Your simulator should be able to execute arbitrary instances of your metamodel.

Simulating Turing Machines

Right, time to write some simulation code. Go grab a cup of coffee and start working through the following steps:

1. Add a new class to one of your plug-ins named TuringHeadController. Make sure it implements the IHeadController interface we defined in the second tutorial and register it with the appropriate extension point.
2. Add a new method `initialize()` to your controller that loads a Turing Machine model from a fixed path.
3. Implement the `nextCommand()` and `reset()` methods:
    - You can access all references and attributes of model elements using generated getter methods.
    - `reset()` selects a state that is marked as being an initial state of the Turing Machine and saves it as the current state in a private field of the controller. (You did think about modeling initial states, right? If not, don't be frustrated, that's not too big of a deal. Just make the necessary changes to your metamodel and regenerate the code.)
    - `nextCommand()` analyzes the outgoing transitions of the currently active state and takes the first one that matches the current character. It then selects the target state of that transition as the new active state and returns the actions of the transition as `HeadCommand`. If there is no transition that matches the current input, return a command that does nothing.
    - At the beginning of `nextCommand()`, check if there is an active state. If not, call `initialize()` and `reset()` before doing the simulation.
    - Remember that if you add the new controller to the `de.cau.cs.rtprak.login.simple` plug-in, you will have to add a dependency to the `...turingmodel` plug-in and make sure the latter exports the required packages.

Testing Your Head Controller

It's time to test the head controller. Here's one way you can go about it:

1. Create a Turing Machine model using the tree editor. Assuming that the initial head position is 1, the machine shall copy the input text infinitely often. That is, if the tape initially contains the word "hello", the machine should generate "hellohellohellohe..." You might remember this task from the second tutorial. To avoid an explosion of the number of states, select a rather small input alphabet for your machine, e.g. h, e, l, and o.
2. Save the model to the fixed path defined in your controller.
3. Select the new head controller in the *Tape* view and test it with input from your editor.

EMF Notifications

We won't touch upon EMF's notification mechanism in this tutorial, but we still wanted to mention it. EMF models can be (and are) used as the main models holding the data edited by applications. The notification mechanisms allow you to add observers to the model that get notified upon a definable set of editing operations executed on the model. Feel free so search the Internet for tutorials and introductions to this topic.